CIS 343 – Project

The Game of Life

Purpose

We have been learning about context in class. In C programming, context is of the utmost importance. For instance, sometimes we may be able to view a `char**` as a multi-dimensional array, and other times we may only be able to view it as a pointer to a pointer. Context affects how we can then use this data structure. This assignment is intended to help you begin to understand the effects of context in C programming, particularly as it relates to the use of pointers.

History

Conway's Game of Life is a cellular automaton created by John Conway, a British mathematician. The "game" is really not a game in the classical sense - there are no win or loss conditions. Instead, the "player" sets the beginning state of cells before allowing the game to evolve to new states through subsequent generations. The game is played on a grid. Each cell of the grid has two states, either off or on, corresponding to dead or alive. The rules are:

- A live cell with less than two live neighbors dies
- A live cell with two or three live neighbors lives
- A live cell with more than three neighbors dies
- A dead cell with three live neighbors becomes live

A "generation" consists of these rules being applied to every cell of the current board state, to create a new board.

Assignment

Your job is to create the Game of Life in C. The initial state of the game will be read in from a file. The file must adhere to the following convention:

- The first byte is the length of the grid
- The second byte is the width of the grid
- The remaining bytes are only 0s or 1s and correspond to a dead or live cell

This means that for an M x N size grid the size of the file should be (M x N) + 2 bytes. So that the grid may be printed on a terminal screen we will limit our size to 30x30. The file name must be passed to the program as a command-line parameter, such that the program will be started with the syntax:

```
./life starter_file
```

Once the program has started, the board from the starter file will be read. If that is not possible the program should print a message and exit.

If the file is valid, the current state of the board will be displayed followed by a menu with the following options:

- Iterate once - will automatically iterate one generation.
- Iterate multiple - will ask the user how many generations and iterate each of those, displaying each iteration.
- Write current state – will ask for a file name and write the current board state to a file. Errors will be reported. If the file cannot be written the program will recover and continue.
- Quit – will gracefully exit, freeing any memory allocated.

Any user errors should be caught, and a message provided that informs the user of correct usage.

Structure

The program must be divided into three parts.

1. A generic file library – Files exist as a collection of bytes on a disk (not characters). You will need to read and write files in C in several classes at GV; this code should be generic enough to save *any* byte array to disk. The code will therefore not be specific to saving and reading a "grid", and conversions may be necessary depending on how you choose to write your board data structure.
2. A library of common functions used in Life – Board creation and printing functions, mutation function, and (possibly) other helper functions.
3. A main file – This file will provide the core logic of the Life game, making use of the functions from the other libraries.

The following files with the following functions must exist. Functions in **bold text** need to be completed, the others will be provided to you. The functions MAY NOT BE MODIFIED (though additional functions may be added if needed):

life.h

```
#ifndef          __H_LIFE__
#define          __H_LIFE__

char** get_grid(int x, int y);
void print_grid(int x, int y, char** grid);
```

```
char** mutate(int x, int y, char** grid);
int get_neighbors(int i, int j, int x, int y, char** grid);

#endif
```

life.c

Implementation of above functions such that:

`char** get_grid(int x, int y)` – Creates a new x by y grid, allocating all memory through malloc. Checks for errors and exits gracefully with appropriate return message if memory cannot be allocated. Returns the grid.

`void print_grid(int x, int y, char** grid)` – Prints a human-readable grid.

`char** mutate(int x, int y, char** grid)` – Given a grid, returns a new grid with Conway's rules applied to each cell. This function will need to create a new grid; the old grid should not be modified while being examined.

`int get_neighbors(int i, int j, int x, int y, char** grid)` – Given a coordinate (i,j) and a grid, determine how many neighbors the cell has. Return -1 if the cell can't exist in the grid. Correctly handle grid boundaries.

file_utilies.h

```
#ifndef         __H_FILE_LIBRARY__
#define         __H_FILE_LIBRARY__

#include <stdio.h>

size_t read_file(char* file_name, char** contents);
size_t write_file(char* file_name, char* contents, size_t size);

#endif
```

file_utilities.c

Implementation of above functions such that:

`size_t read_file(char* file_name, char** contents)` – Reads in the bytes of a file into an array of character bytes pointed to by `contents`. Uses malloc to create the memory for the string. Returns the number of bytes read. The calling function must have access to all data read in via the variable passed to `contents`.

`size_t write_file(char* file_name, char* contents, size_t size)` – Writes a byte array to the file name provided. Note, the signature is correct.

`main.c`

Main program logic. This entire file is given to you.

Tips

- A file is just a stream of bytes – NOT just printable characters. Be sure you understand the difference.
- Though the C standard does not define the size of C data types (instead it merely provides sizes in relation to other data structures), we can safely use the `char` to store a byte.
- A `char**` is a pointer to a pointer, but because an array in C is just a pointer we can simulate a grid with it as well. We simply must be careful about the memory allocation. A `char**` could also be the address of a pointer though…

Grading Criteria

Per the syllabus, programs that don't compile receive no credit.

Compiles with no warnings	10 points
Adheres to style guide	10 points
All files and functions provided	20 points
Properly handles errors	10 points
Reads files correctly	15 points
Writes files correctly	15 points
Correct program logic	20 points
Total	100 points

Submission

The project must be zipped or tarred and uploaded to Blackboard before the due date.